Why Johnny Can't Program

Neville Holmes, University of Tasmania

Although written nearly a year ago, this column quite neatly follows last month's, in which I charged that misguided attitudes to programming restrict the layout of program code too much. This month, I claim that programmers are not educated well enough to code really good programs.

Johnny's problems were highlighted for me by Bertrand Meyer's December 1999 Component and Object Technology column ("A Really Good Idea," *Computer*, pp. 144-147), which I read for two reasons. First, I've long been bemused by the prevalence of Objects. Never having had much to do with them, I've found that what I read about Objects either passed over my head or resembled a strictly applied version of what I knew long ago as modular programming.

Second, I wanted to know why Meyer, an authority on the topic, thought Objects such a good idea. Although Meyer explained why in his column, most entertainingly and eloquently, I still felt confused about the nature of the really good idea.

One really good idea seemed to be about software construction in the large: "…when it comes to building complex, evolutionary, mission-critical systems, OO solutions are our best bet. Nothing else has come to challenge them." This passage bespeaks an emphasis on technical management. Indeed, there's nothing like the thorough and controlled application of a clearly specified and well-tested technology to ensure a complex technical project's prosperity. So, as a basis for good technical management, Object Technology (or is it Component Technology?) should be a really good idea.

The other really good idea seemed to be about what developers once called systems analysis, the arena of the promoted programmers who specified the programs and modules to be coded by their less-experienced or less-talented workmates. Meyer gave as examples two general principles, although *objectives* might be a better word:

- "the OO view [is] that we are building little machines, each with its official control panel . . . serving as the obligatory path to the internals," and
- "to keep modules independent from each other's implementation decisions and hence from variations in each other's implementations."

Had these examples of objectives appeared early in the column, I probably would have dismissed the piece as another case of modular programming revisited. But by the time I got to them, I had slogged through two rather disturbing programming exhibits. True, Meyer offered these exhibits as examples of bad programming. My only quarrel with him is that he didn't show how really bad they are.

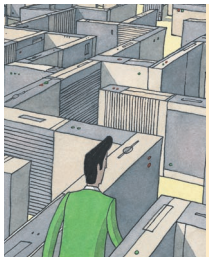
His failure to do so caused me to conclude that OO technology focuses on technical management and systems analysis, and away from program coding. Methodology has drowned the craft. Johnny may be a software engineer now, but he can't program anymore—at least not properly.

Why can't Johnny program? Meyer's two programming exhibits suggest several reasons.

JOHNNY IS INNUMERATE

Evidence for asserting that Johnny is innumerate lies in the pivot "solution" to the main Y2K problem, referred to in Meyer's column as the "windowing Y2K technique." He properly lambastes it, then later asserts that the solution lies in the use of information hiding.

That might be so at the systems level, or even at the coding level, but Meyer's criticism overlooks that the pivot technique is simply an innumerate corruption of a perfectly good programming technique that uses what school mathematics curricula call "clock arithmetic."



Innumerate, illiterate, and overwhelmed, today's professionals are torn between system design and program coding.

To keep the explanation simple, let's forget about minutes and suppose that in a program we must deal with a 24-hour clock precise to the nearest hour; thus our code must deal with integers in the range of 0 to 23. Such clocks mainly allow us to compute periods of time. In this example, our starting time and our finishing time will be integers in the range of 0 to 23.

The program computes the elapsed time by subtracting the starting time from the finishing time. What happens if the computed elapsed time turns out to be negative? Such a result indicates that the period of time included midnight. No problem in clock arithmetic—you just add a modulus of 24, with no pivot necessary.

The 24-hour clock compares directly to double-digit calendar-year encoding, but with a modulus of 100. Clock arithmetic, as taught in elementary school and

Continued on page 158

Continued from page 160

```
//function to eliminate blanks from a string
  void eatspaces (char * str) {
    int i=1; /* copy to offset within string */
    int j=1; /* copy from offset within string */
    while ((*(str + i) = *(str + j++)) !='\0')
      if (*(str + i) != ' ') i++;
    return;
  }
```

Figure 1. Coding example with illiterate comments.

```
//function to eliminate blanks from a string
  void eatspaces (char * string) {
    int to = 1;
    int from = 1;
    while ((*(string + to) =
            *(string + from++)) != '\0')
          if (*(string + to) != ' ') to++;
    return;
  }
```

Figure 2. The code of Figure 1 with explanatory data names added.

as known and used by skilled programmers for at least 40 years, easily and correctly deals with such encoding. Therefore, innumerate Johnny encoded all programs with this particular Y2K bug. Johnny also wrote the pivot patches.

JOHNNY IS ILLITERATE

Meyer's second programming example—drawn "from one of the most frequently used C++ introductory textbooks" and shown in Figure 1—supports the claim that Johnny is illiterate.

Meyer says of this example that it "can't have anything to do with object technology" and proceeds to spell out why. What disturbs me about the example is not the points Meyer makes against the algorithm, which I heartily agree with, but the point he *doesn't* make about the coding, at least not directly: That the code is so bad that he must explain it in bitter detail speaks volumes. Good code is self-explanatory, at least to anyone familiar with the coding system used.

Why is the example's code so bad? The comments provide one clue. Comments—as opposed to *remarks*, which address aspects external to the code—explain the code itself. To quote a saying that once circulated among programmers: "A comment is a confession of failure."

The most significant aspect of a program is the names given to its data—a good programmer will put great effort into choosing names that explain the data. This practice makes expressions that use the names much more meaningful, and should make comments unnecessary. Let's try it—Figure 2 shows the results.

If anything needs to be explained now, it's the details specific to the C++ coding system; we no longer need the original comments. This matter is not trivial. Good programming practice focuses on functional coding, not on decoration with comments. Developers should design code, at least public code, so that it can be understood.

Johnny shows himself to be illiterate in more than his choice of names. Problems crop up in smaller details, too. Compare the code of Figure 2 with that of Figure 3.

There's only a small difference—the conditions of the `while` and the `if` have been rearranged internally. We read this kind of code from left to right, putting together the meaning as we go. The meaning of both the `while` and the `if` hinge on the comparison `!=`. These meanings become more obvious in the rearranged example, first because the comparisons come up early, where you see them before your attention starts to wander, and second because you don't need to store a complex comparand in your working memory while waiting for the comparison to appear.

This code needs further literate improvements. Most conspicuously, the blank and end-of-string characters should be declared and everywhere used as named constants.

```
//function to eliminate blanks from a string
  void eatspaces (char * string) {
    int to = 1;
    int from = 1;
    while ('\0' != (*(string + to) =
                    *(string + from++)))
          if (' ' != *(string + to)) to++;
    return;
  }
```

Figure 3. The code of Figure 2 with reversed comparisons.

These may seem minute details, but, as Meyer says, "The in-the-large aspects of programming rely on the lower-level parts, and you can't get them right unless you get the small things right too."

Listing Johnny's inadequacies does not, however, reveal why Johnny got that way. It's too simple merely to accept that the community as a whole is getting more illiterate and more innumerate. Presumably, Johnny received training. So why didn't it make him numerate and literate?

JOHNNY IS OVERWHELMED

I believe that Johnny, trained as a software engineer and employed to develop software systems, has bitten off more than he can comfortably chew. Meyer borrows a metaphor from Isaiah Berlin to tell us that object technology has "a little of the fox and a little of the hedgehog." Object technology, Meyer insists, requires that Johnny be competent both in the small, like the fox, and in the large, like the hedgehog.

But consider this point: In other branches of engineering, the professional engineers deal with the in-the-large tasks while the tradespeople deal with the in-the-small ones. Each group has its particular training, skills, and duties. With so much to learn and know and do in an endeavor such as building a highway or a ship, a single profession or calling cannot skillfully perform all the work. Yet we blithely insist that software engineers must be both foxes and hedgehogs.

The best way to build Meyer's "complex, evolutionary, mission-critical systems" involves creating a team that consists of both foxes and hedgehogs, rather than a single group that vainly strives to acquire the traits of both. Such a team will consist of

- system engineers with in-the-large skills like, but not limited to, those Meyer describes, and
- programmers with in-the-small skills that include and lie behind those he describes.

I deliberately use "system engineer" rather than "software engineer." The professional programmer must know and understand clock arithmetic, but the system engineer must be responsible for determining whether a mere 24-hour clock provides a safe system solution. I don't think that "software engineer" describes this role well.

In real life, engineers should be designing and validating the system, not the software. If you forget the system you're building, the software will often be useless.

Computing professionals must know vastly more now than they needed to know 40 years ago, but even then the field often distinguished between programmers and systems analysts. Meanwhile, Johnny must know too much to be at the same time a skilled programmer and a skilled system engineer. That's why he's overwhelmed.

But Johnny's problem goes deeper: If he wants to be a professional programmer, he must learn on the job. Programmers have few trade courses to select from, many fewer than those available to carpenters, plumbers, and electricians. Further, very few professional courses will assist Johnny if he wants to become a straightforward, generalist, humanist, system engineer. Specialists such as computer system engineers and software engineers appear to dominate the field.

No wonder Johnny has a problem. Let's move to give him the proper training to become whichever he wants to be—programmer or system engineer. ✸

Neville Holmes *is a lecturer under contract at the University of Tasmania's School of Computing. Contact him at neville.holmes@utas.edu.au.*

Circulation: *Computer* (ISSN 0018-9162) is published monthly by the IEEE Computer Society. **IEEE Headquarters**, Three Park Avenue, 17th Floor, New York, NY 10016-5997; **IEEE Computer Society Publications Office**, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314; voice (714) 821-8380; fax (714) 821-4010; **IEEE Computer Society Headquarters**, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. IEEE Computer Society membership includes $13 for subscription of *Computer* magazine ($13 for students). Nonmember subscription rate available upon request. Single-copy prices: members $10.00; nonmembers $20.00. This magazine is also available in microfiche form.

Postmaster: Send undelivered copies and address changes to *Computer*, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08855. Periodicals Postage Paid at New York, New York, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail (Canadian Distribution) Agreement Number 0487910. Printed in USA.

Editorial: Unless otherwise stated, bylined articles, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *Computer* does not necessarily constitute endorsement by the IEEE or the Computer Society. All submissions are subject to editing for style, clarity, and space.

COMPUTER
Innovative technology for computer professionals